Allocating memory arrays for polyhedra

Doran K. Wilde, Sanjay Rajopadhye

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Allocating Memory Arrays for Polyhedra

Doran K. Wilde and Sanjay Rajopadhye

N° 2059

Juillet 1993

PROGRAMME 1

Architectures parallèles,
bases de données,
réseaux et systèmes distribués

ISSN 0249-6399

Allocating Memory Arrays for Polyhedra

Doran K. Wilde and Sanjay Rajopadhye*

Programme 1 — Architectures parallèles, bases de données, réseaux
et systèmes distribués
Projet API

Rapport de recherche n ° 2059 — Juillet 1993 — 24 pages

Abstract: We have been investigating problems which arise in compiling single assignment languages (in which memory is not explicitly allocated) into parallel code. Like standard parallelizing compilers, different index space transformations are performed on variables declared over convex polyhedral regions. Polyhedra can be transformed in such a way as to reduce the volume of the bounding box which we use to reduce the amount of memory allocated to a variable. Allocation of memory to variables which are defined over finite convex polyhedral regions requires a tradeoff in the complexity of the memory addressing function versus the amount of memory used. We present a tradeoff in which the memory address function is limited to an affine function of the indices (thus memory is allocated to a rectangular parallelepiped region). Given this constraint, we seek a unimodular transformation which minimizes the volume of the bounding box of the polyhedron. This is a non-linear programming problem. We present a method in which the volume of the bounding box is minimized one dimension at a time by a succession of skewing transformations. Each one of these is a linear programming problem.

Key-words: Memory allocation, Polyhedra, Linear programming

(Résumé : tsvp)

*This work was partially supported by the Esprit Basic Research Action NANA 2, Number 6632

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : (33) 99 84 71 00 – Télécopie : (33) 99 38 38 32

Allocation de Mémoire pour des Polyèdres

Résumé : L'évaluation des équations récurrentes linéaires est plus efficace s'il y a stockage de la mémoire assignée aux variables. L'affectation de la mémoire aux variables qui sont basées sur les domaines convexes demande un compromis entre la complexité de la fonction qui trouve un élément d'une variable en mémoire, et la quantité de mémoire affectée au stockage de cette variable. Cet article propose un compromis dans lequel la taille de le mémoire affecté est minimisé sous contrainte que la fonction pour adresser le mémoire soit affine.

Mots-clé : Allocation de Mémoire, Polyèdres, Programmation Linéaire

1 INTRODUCTION

In this paper we discuss an issue that arises when considering the simulation or evaluation of systems of affine recurrence equations (SARE) [11]. Here, we limit our SARE's to equations involving variables defined over *finite* convex polyhedral domains. We have been working with the ALPHA environment [10, 9] which is based on the formalism of affine recurrence equations. ALPHA is similar to the CRYSTAL language [6] in that is was designed to facilitate space-time transformations of programs, but is less ambitious than CRYSTAL, and is based on the substitution principle which allows program transformations to be done through syntatic rewriting. Systems of affine recurrence equations are referentially transparent, meaning that an expression has the same meaning in every context and evaluation of a recurrence equation has no side effects. This property is shared by functional languages which makes describing systems with recurrence equations very similar to programming in a functional language. Variables in a such a system are single-assignment, meaning that each variable name is associated with one and only one value. Another side effect of referential transparency is that there is flexibility in the order that equations are evaluated (or reevaluated). As long as flow dependencies are respected, equations may be evaluated, or may be scheduled to be evaluated, in any order—even in parallel. Since variables are all single assignment, there are no write dependencies or anti-dependencies to be concerned about.

In the simulation or execution of systems of affine recurrence equations, there are two strategies involving variables

1. *No variable storage.* Variables are recomputed according to their functional definitions each time they are needed. This is called *demand driven evaluation* and may be implemented by graph reduction [12].

2. *Variable storage.* The first time a variable is computed, the result is stored. The single assignment property of systems of recurrence equations ensures that each variable will only ever have one value. If the value of a variable is ever needed a second or subsequent time, its value is not recomputed, but is obtained from memory. This technique is known as *applicative caching* or *tabulation.* [3, 8]

This paper will deal with ways to facilitate the second strategy by statically allocating memory for variables declared over arbitrary finite convex polyhedral domains.

Example. Fibonacci Series

$$F ::= \{ F_n : F_n \in integer,\ 1 \leq n \}$$
$$F_n = \begin{cases} 1 & n = 1 \\ 1 & n = 2 \\ F_{n-1} + F_{n-2} & n > 2 \end{cases}$$

To compute F_n without using variable storage requires exponential time. However, if variable storage is allocated to store values of variable F when they are computed, then computation requires only linear time.

In a system with variable storage, each variable element is computed only once, minimizing computation time at the expense of memory allocation. Methods of allocating memory for arrays can be divided into static and dynamic methods. Static methods declare the allocation of memory for an entire variable array at the beginning of a program. Static methods include the use of pointer arrays, array linearization techniques, and static caching (replacement schedule is precomputed) . In contrast, dynamic methods allocate memory for elements, or groups of elements at run time, when they are computed. Variables remain in memory until they are deallocated or replaced. Dynamic methods include hashing and caching (with a dynamic replacement algorithm). Infinite domains must be allocated dynamically, since they can never exist in memory in their totality at one time. Static and dynamic methods can be used together.

This paper investigates a static method of variable storage in which a variable based on a convex polyhedron is transformed in such a way as to reduce the volume of the bounding box containing the variable. This reduces the amount of memory that needs to be allocated to the variable. In section 2, we present some background information and definitions referred to in the paper. In section 3, we will show that allocation of memory to variables which are defined over finite convex polyhedral regions requires a tradeoff in the complexity of the memory addressing function versus the amount of memory used. We present a tradeoff in which the memory address function

is limited to an affine function of the indices and show that this requires that memory be allocated to a rectangular parallelepiped region of variable index space. In section 4, we discuss how bounding boxes are computed and show that a unimodular transformation can be used to reduce the volume of the bounding box of a polyhedron without changing the complexity of the addressing function. We show how this can be done using a sequence of skews and how those skews can be computed. We formalize this as a non-linear programming problem. In section 5, we present an algorithm in which the volume of the bounding box is minimized one dimension at a time by a succession of skewing transformations. Each one of these is a linear programming problem. In section 6, we discuss some techniques of scanning variable spaces allocated by our method and in section 7, we present our conclusions.

2 Review and Definitions

2.1 Review of dual representations of polyhedra

Any convex polyhedron $\mathcal{P}$ can be represented implicity using a homogeneous system of inequalities

$$\mathcal{P} = \{x \ : \ A \begin{pmatrix} \lambda x \\ \lambda \end{pmatrix} \geq 0, \quad \lambda > 0\} \tag{1}$$

where A is a constant integer matrix in which each row of A represents one inequality constraining the domain, x is a rational vector and λ is an integer scalar such that λx is an integer vector. The set of rational points x which satisfy the above system of constraints form a convex polyhedron $\mathcal{P}$. Equation 1 is called the *implicit* representation of $\mathcal{P}$. The polyhedron $\mathcal{P}$ also has an equivalent dual *parametric* representation

$$\mathcal{P} = \{x \ : \ \begin{pmatrix} \lambda x \\ \lambda \end{pmatrix} = R\mu, \quad \mu \geq 0, \quad \lambda > 0\} \tag{2}$$

stated in terms of the vertices (columns of integer matrix R) of the polyhedron, where μ is a free valued rational column vector under the positivity restriction specified above. Procedures exist to compute the dual representations of $\mathcal{P}$, that is, given A, compute R, and visa versa [7]. We have such

a procedure in the geometric function library used by the ALPHA system. Once R is determined, any non-negative linear combination of its columns is a feasible solution to the system of constraints given in equation 1 and yeilds a point somewhere inside $\mathcal{P}$. Each column of R is of the form $\begin{pmatrix} \xi r \\ \xi \end{pmatrix}$ where $\xi \geq 0$. If $\xi > 0$, that column is called a *vertex*, and if $\xi = 0$, it is called a *ray*. A point(ray) in $\mathcal{P}$ is an *extreme point(ray)* if and only if it cannot be expressed as a positive combination of other points and rays of $\mathcal{P}$. If all columns of R are vertices, the polyhedron is bounded, has a finite volume, and is called a *polytope*.

2.2 Domains and Variables

Whereas a polyhedron is a region containing an infinite number of rational points, a *domain*, as we use the term in this paper, only refers to the lattice of integral points in a polyhedron.

Definition 1 *A finite polyhedral* domain *of dimension n is defined as*

$$\mathcal{D} : \{i \in \mathcal{Z}^n, i \in \mathcal{P}\} = \mathcal{Z}^n \cap \mathcal{P} \tag{3}$$

where $\mathcal{P}$ is a finite convex polyhedron of dimension n as defined in equations 1 and 2.

A domain consisting of a lattice of points inside a bounded polyhedron (or polytope) will contain a finite number of points and is called a *finite domain*. Given a domain $\mathcal{D}$, we now define two functions as follows :

Definition 2 *The* volume function*, volume : $\mathcal{D} \rightarrow \mathcal{Z}$, is defined to be the number of distinct points in the domain.*

Definition 3 *A* lexical mapping function*, lex : $\mathcal{Z}^n \rightarrow \mathcal{Z}$, maps each point in a domain $\mathcal{D}$ to a unique integer in the set $\{n : n \in \mathcal{Z}, 0 \leq n < volume(\mathcal{D})\}$. This function is not necessarily unique.*

In affine recurrence equations of the type considered here and in the language ALPHA, every variable is declared over a convex polyhedral domain. Here, we formalize the definition of a variable.

Definition 4 *A* variable X *of type* datatype *declared over a domain* $\mathcal{D}$ *is defined as*

$$X ::= \{\ X_i\ :\ X_i \in \text{datatype},\ i \in \mathcal{D}\ \} \tag{4}$$

where X_i *is the element of* X *corresponding to the point* i *in domain* $\mathcal{D}$.

The memory allocated for a variable X is a vector of elements of length $volume(X.\mathcal{D})$ starting at some address *base*. The amount of memory allocated for variable X is defined in equation 4 is

$$size\ of(X) = size\ of(X.datatype) \times volume(X.\mathcal{D}) \tag{5}$$

and the address of an element X_i of variable X is:

$$address(X_i) = base + size\ of(X.datatype) \times lex(i) \tag{6}$$

where $volume(D)$ and $lex(i)$ are as previously described.

In the next section, we will present the *lex()* functions for rectangular and triangular domain based variables. In sections 4.2 and 4.3, we will restrict the complexity of the function *lex()* to be an affine function of the indices of the domain, and then minimize the amount of memory allocation needed under that constraint.

3 Complexity of lexical functions

The lexical mapping function $lex : \mathcal{Z}^n \rightarrow \mathcal{Z}$ maps each index point in n-space to a unique integer and hence is an ordering function, similar to other functions which specify an order that points in a n-dimension space are to be read (or processed in some other way). Our ordering is simply the order in which elements of a variable are stored sequentially in memory, and for now we will assume that the order is unconstrained, that is any ordering is just as good any another. In section 6.1, we will discuss the implication of a partial ordering constraint (i.e. a timing function) being imposed on the variable.

Since, during the course of a simulation or evalution of a system of recurrence equations, the *address()* function has to be evaluated every time a variable is accessed, and since *lex()* is used in the function *address()*, its complexity becomes of concern. If the complexity of computing the position

of a variable element in memory exceeds the complexity of the actual computation, then the expected advantage of using memory to store variable results is lost.

It will be useful at this point to take a look at two special classes of domains. The first type is the rectangular parallelepiped domain which has affine *lex()* functions. The second type is the triangular domain which has polynomial complexity *lex()* functions. It will be seen from these two examples how the complexity of the *lex()* function relates to the shape of the domain that it enumerates.

3.1 Example: lexical functions for rectangular domains

Rectangular domains are a class of domains that can be scanned by a set of nested DO-loops with *constant* lower and upper bounds. This class of domains is interesting because lexical functions for rectangular domains are affine functions of their indices.

This class of domains has the following form:

$$\begin{aligned} \mathcal{D} &= \{x \ : \ x \in \mathcal{Z}^n, \quad x_1 \geq l_1, \quad x_1 \leq u_1 \\ &\qquad\qquad\qquad\quad x_2 \geq l_2, \quad x_2 \leq u_2 \\ &\qquad\qquad\qquad\qquad\quad \vdots \quad \vdots \\ &\qquad\qquad\qquad\quad x_n \geq l_n, \quad x_n \leq u_n \ \} \\ &= \{x \ : \ x \in \mathcal{Z}^n, \qquad l \leq x \quad \leq u\} \end{aligned} \tag{7}$$

where x_i is the i^{th} index (coordinate) of the point x, and where l_i and u_i are constant scalars for $i = 1 \cdots n$, with n being the dimension of the domain.

One can define *lex* and *volume* functions for a rectangular domain in terms of a vector w whose elements w_i are defined recursively in terms of l_i and u_i is defined as:

$$w_i = \begin{cases} 1 & \text{when} \quad i = 1 \\ w_{i-1}(u_{i-1} - l_{i-1} + 1) & \text{when} \quad 1 < i \leq n+1 \end{cases} \tag{8}$$

The scalar w_i is the volume of the domain in the first $i-1$ dimensions. The *lex* and *volume* functions are then defined in terms of w:

$$lex(i) \quad = \quad \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{pmatrix} \circ (i - l) \tag{9}$$

$$volume(X.\mathcal{D}) \quad = \quad w_{n+1} \tag{10}$$

where n is the dimension of $X.\mathcal{D}$ and the operation $\circ$ is a vector dot product. The w-vector above is called the doping vector in compiler terminology [1].

Figure 1 shows programs for rectangular domains of dimensions 1, 2, and 3 with their associated functions. It can be observed from the examples that lexical functions for this kind of domain are always affine functions of the indices.

Nested Loop Program	Lexical and Volume Functions
DO $i = L_i \ldots U_i$ { process x_i }	$lex(i) = i - L_i$ $volume(i) = (U_i - L_i + 1)$
DO $i = L_i \ldots U_i$ DO $j = L_j \ldots U_j$ { process x_{ij} }	$lex(i,j) = (U_j - L_j + 1)(i - L_i) +$ $j - L_j$ $volume(i,j) = (U_j - L_j + 1)(U_i - L_i + 1)$
DO $i = L_i \ldots U_i$ DO $j = L_j \ldots U_j$ DO $k = L_k \ldots U_k$ { process x_{ijk} }	$lex(i,j,k) = (U_k - L_k + 1)(U_j - L_j + 1)(i - L_i) +$ $(U_k - L_k + 1)(j - L_j) +$ $k - L_k$ $volume(i,j,k) = (U_k - L_k + 1)(U_j - L_j + 1)(U_i - L_i + 1)$

Figure 1: Lexical functions for Rectangular Domains

3.2 Example: lexical functions for triangular domains

In this section, the class of triangular domains is investigated. This class of domains can be scanned by a set of nested DO-loops whose lower and upper bounds are affine functions of outer nested loop indicies. The lexical scanning functions for this class of domains are polynomial functions of the indices. Triangular domains are also important because all convex domains can always be subdivided into triangular sub-domains. A similar method was employed in [4] in which scanning functions were found for arbitrary domains by subdividing them into trapazoidal sub-domains. Since the *lex()* function of any convex domain can be expressed as a combination of the

lex() functions of its triangular sub-domains combined with IF-statements, the complexity of the *lex()* function of any convex domain is of the same order as the complexity of triangular domains.

This class of domains has the following general form:

$$\begin{aligned}\mathcal{D} = \{x \ : \ x \in \mathcal{Z}^n, \quad & x_1 \geq b_1, && x_1 \leq d_1 \\ & x_2 \geq b_2 + a_{21}x_1, && x_2 \leq d_2 + c_{21}x_1 \\ & x_3 \geq b_3 + a_{31}x_1 + a_{32}x_2, && x_3 \leq d_3 + c_{31}x_1 + c_{32}x_2 \\ & \vdots && \vdots \\ & x_n \geq L_n(x_1, \cdots, x_{n-1}) && x_n \leq U_n(x_1, \cdots, x_{n-1}) \ \}\end{aligned} \tag{11}$$

where x_i is the i^{th} index (coordinate) of the point x, where a_{ij}, b_i, c_{ij}, and d_i are all constant valued coefficients, and where the scalar functions $L_i()$ and $U_i()$ are the respective lower and upper bounds of the index x_i for each dimension i, and are affine functions of lower dimensioned indices $x_1, x_2, \cdots, x_{i-1}$. Figure 2 shows programs for processing all elements in triangular domains of dimensions 1, 2, and 3. The associated lexical functions produce an enumeration number for each element in $\mathcal{D}$ corresponding to the order in which elements of $\mathcal{D}$ are processed (with 0 being the first). The complexities of

Nested Loop Program	Lexical Function
DO $i = L_i \ldots U_i$ { process x_i }	$lex(i) = i - L_i$
DO $i = L_i \ldots U_i$ DO $j = L_j(i) \ldots U_j(i)$ { process x_{ij} }	$lex(i,j) = \sum_{a=L_i}^{i-1} [U_j(a) - L_j(a)] + j - L_j(i)$
DO $i = L_i \ldots U_i$ DO $j = L_j(i) \ldots U_j(i)$ DO $k = L_k(i,j) \ldots U_k(i,j)$ { process x_{ijk} }	$lex(i,j,k) = \sum_{a=L_i}^{i-1} \sum_{b=L_j(a)}^{U_j(a)} [U_k(a,b) - L_k(a,b)] + \sum_{b=L_j(i)}^{j-1} [U_k(i,b) - L_k(i,b)] + k - L_k(i,j)$

Figure 2: Lexical functions for Triangular Domains

the lexical functions in figure 2 are polynomials of the same degree as the dimension of the respective domains.

3.3 Limiting the complexity of lexical functions

Complex lexical functions result in complex addressing functions for variables allocated in memory (equation 6). Since the addressing function must be called each time a variable is accessed, the complexity of the entire simulation is affected. It is therefore very desirable to limit, as much as is possible, the complexity of the lexical function.

To reduce the complexity of the lexical function for a domain, more memory can be allocated than is actually needed to represent the domain (more than the volume of the domain). If an amount of memory is allocated to store the smallest rectangular domain which contains the domain of the variable being allocated (which we will call the *bounding box* from now on) then the lexical function becomes an affine function of the indices. The cost for the simplified lexical function is that more memory is allocated than is necessary[1]. The difference in the volume of the bounding box domain and the volume of the variable domain is the amount of unusable memory that needs to be allocated in order to have the benefits of an affine lexical function. In this paper, we propose to limit the complexity of the lexical function to an affine function and then to try to minimize the amount of memory which is allocated.

4 Addressing and Volume Functions for Bounding Boxes

In section 3.1, we presented the *lex()* and *volume()* functions for rectangular domains. Since the *lex()* function of a bounding box is an affine function, the *address()* function which is based on it (equation 6) is also affine. We will show how to compute a transformation T which alters the shape of a domain $\mathcal{D}$, such that the volume of the bounding box of the transformed domain $\mathcal{D}'$

[1]The amount of memory allocated to a bounding box is limited in the worst case to the volume of the domain times $n!$ where n is the dimension.

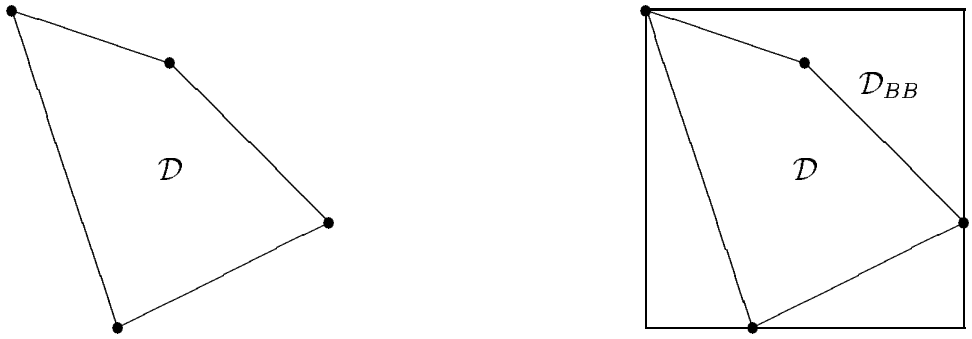

Figure 3: Domain $\mathcal{D}$ and its Bounding Box

is reduced. It will be shown that affine *lex()* functions are closed under affine unimodular transformations of the underlying domain.

4.1 Use of bounding boxes

The figure 3 shows (on the left) a domain $\mathcal{D}$ and (on the right) $\mathcal{D}$ inside its bounding box $\mathcal{D}_{BB}$. In our approach, to represent $\mathcal{D}$ in memory, we in fact allocate enough memory to represent $\mathcal{D}_{BB}$, a superset of $\mathcal{D}$ allowing the use of an affine function to address an element of $\mathcal{D}$ in memory (equations 6 and 9). The area outside of $\mathcal{D}$ but inside $\mathcal{D}_{BB}$ represents area which is wasted memory. The amount of waste is proportional to $volume(\mathcal{D}_{BB}) - volume(\mathcal{D})$.

A *bounding box* of $\mathcal{D}$ is defined as the smallest rectangular domain $\mathcal{D}_{BB}$ which contains $\mathcal{D}$. As defined in equation 7, rectangular domains are characterized in terms of constant vectors u and l. We will show how the vectors u and l of a bounding box are computed for a given finite domain $\mathcal{D}$. $\mathcal{D}$ is defined implicity and parametrically as follows:

$$\mathcal{D} = \{x : x \in \mathcal{Z}^n, \quad A \begin{pmatrix} \lambda x \\ \lambda \end{pmatrix} \geq 0, \quad \lambda > 0\} \tag{12}$$

$$= \{x : x \in \mathcal{Z}^n, \begin{pmatrix} \lambda x \\ \lambda \end{pmatrix} = R\mu, \quad \mu \geq 0, \quad \lambda > 0\} \tag{13}$$

where A and R are constant integer matrices, x and μ are rational vectors, and λ is an integer scalar such that λx is an integer vector.

The columns of the R matrix represent the k extremal vertices of the polyhedron. To bound a finite polyhedron, it is sufficient to bound its extremal

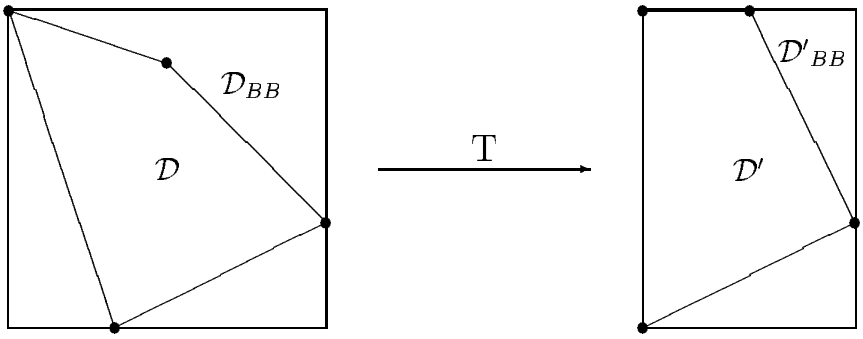

Figure 4: Transformed Domain $\mathcal{D}'$ and its Bounding Box

vertices. The rational constants l_i and u_i are therefore defined as the minima and maxima, respectively, of the k values r_{ij}'s in the i^{th} row of R:

$$\begin{aligned} l_i &= \min(r_{i1}, r_{i2}, \cdots, r_{ik}), \quad 1 \leq i \leq n \\ u_i &= \max(r_{i1}, r_{i2}, \cdots, r_{ik}), \quad 1 \leq i \leq n \end{aligned} \tag{14}$$

The bounding box $\mathcal{D}_B$ is defined in terms of l_i and u_i.

$$\mathcal{D}_{BB} = \{x \ : \ x \in \mathcal{Z}^n, \ l_i \leq x_i \leq u_i, \ 1 \leq i \leq n\} \quad \supseteq \ \mathcal{D} \tag{15}$$

4.2 Reducing the Memory Allocation

To reduce wasted memory, we try to find a linear unimodular transformation $T : \mathcal{D} \rightarrow \mathcal{D}'$ which changes the shape of the domain D without changing its volume and such that the difference between the transformed domain $\mathcal{D}'$ and its bounding box, i.e. $volume(\mathcal{D}'_{BB}) - volume(\mathcal{D}')$, is minimized. Since the transformation T is unimodular (i.e. $det(T) = 1$) it is volume preserving ($volume(\mathcal{D}') = volume(\mathcal{D})$). Thus, to minimize the waste, it suffices to minimize $volume(\mathcal{D}'_{BB})$.

Figure 4 illustrates domain $\mathcal{D}'$ which is domain $\mathcal{D}$ transformed under some transformation T. The bounding box $\mathcal{D}'_{BB}$ of the transformed domain is smaller than the bounding box of the untransformed domain. Less memory needs to be allocated for the bounding box of $\mathcal{D}'$ than for that of the original domain $\mathcal{D}$.

The problem can now be restated in the following terms:

Given a domain $\mathcal{D}$,
Find an unimodular transformation $T : \mathcal{D} \rightarrow \mathcal{D}'$
such that the $volume(\mathcal{D}'_{BB})$ is minimized.

In terms of D and T, the domain $\mathcal{D}'$ is defined as as:

$$D' = \{x' \ : \ x' = Tx, \quad x \in D\} \tag{16}$$

$$= \{x' \ : \ x' \in \mathcal{Z}^n, \begin{pmatrix} \lambda x' \\ \lambda \end{pmatrix} = TR\mu = R'\mu, \quad \mu \geq 0, \quad \lambda > 0\} \tag{17}$$

where R is a constant integer matrix, T is a unimodular matrix, λ is an integer scalar, and x, x' and μ are rational vectors, and $\lambda x'$ is an integer vector.

Having the matrix $R' = TR$, the vectors l', u' and w' are computed from R' in the same manner that l, u and w were computed from R. The new *lex()* and *volume()* functions of the transformed domain are:

$$lex(x') = \begin{pmatrix} w'_1 \\ w'_2 \\ \vdots \\ w'_n \end{pmatrix} \circ (x' - l') = w' \circ x' - w \circ l' \tag{18}$$

$$volume(D'_{BB}) = w'_{n+1} \tag{19}$$

where n is the dimension of both D' and D'_{BB} and the operation $\circ$ is a vector dot product.

Theorem 1 *The lex function of a transformed domain D' is an affine function of x.*

Proof: Given *lex(x')*, in terms of transformed points x', an affine function *lex(x)* may be derived as follows:

$$\begin{aligned} lex(x') &= w' \circ x' - w' \circ l' \\ &= w' \circ (Tx) - w' \circ l' \\ &= (T^T w') \circ x - w' \circ l' \\ &= lex(x) \end{aligned}$$

Thus, given the transformation matrix T, the constant vectors w' and $T^T w'$ can be computed giving the coefficients for an affine *lex()* function

in terms of points in the original domain. The integer scalar constant $w' \circ l'$ can also be computed. Thus,

$$lex(x) = (T^T w') \circ x - w' \circ l' \tag{20}$$

is affine.
□

Next we show how to find the matrix T to reduce the volume of D'_{BB}.

4.3 Minimizing the volume of the bounding box

In this section we focus on the problem of finding a transformation T which minimizes the volume of the bounding box of $\mathcal{D}'$, where $\mathcal{D}' = \mathcal{D}.T$.

If we define the size of the bounding box in the i^{th} dimension to be:

$$s_i = u_{i-1} - l_{i-1} + 1 \tag{21}$$

then by inspecting equations 8 and 10 we can see that the volume of the bounding box is the product of the lengths of its sides :

$$\begin{aligned} volume(\mathcal{D}_{\mathcal{BB}}) &= w_{n+1} \\ &= s_1 \times s_2 \times \cdots \times s_n \end{aligned} \tag{22}$$

To minimize the volume of the bounding box, we need to choose a transformation T that reduces the lengths of the transformed sides, where T is a unimodular transformation (so that $volume(\mathcal{D}) = volume(\mathcal{D}')$). We now define T to be the product of n unimodular *skews* [14], (one for each of the n dimensions), and an unimodular *scale*:

$$T = S \times T_n \times \cdots \times T_2 \times T_1 \tag{23}$$

where each T_i is a unimodular skew and S is a unimodular scale as follows:

$$T_i = \begin{bmatrix} 1 & \cdots & 0 & 0 & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \vdots & & \vdots \\ 0 & & 1 & 0 & 0 & & 0 \\ t_{1i} & \cdots & t_{(i-1)i} & 1 & t_{(i+1)i} & \cdots & t_{ni} \\ 0 & & 0 & 0 & 1 & & 0 \\ \vdots & & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & 0 & 0 & \cdots & 1 \end{bmatrix} \quad S = \begin{bmatrix} s_1 & \cdots & 0 & 0 & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \vdots & & \vdots \\ 0 & & s_{i-1} & 0 & 0 & & 0 \\ 0 & & 0 & s_i & 0 & & 0 \\ 0 & & 0 & 0 & s_{i+1} & & 0 \\ \vdots & & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & 0 & 0 & \cdots & s_n \end{bmatrix}, \prod_{i=1}^{n} s_i = 1 \tag{24}$$

Theorem 2 *An n-dimensional unimodular matrix X, with the property that the leading diagonal submatrices (square submatrices of X containing the element x_{11}) have non-zero determinants, can be factored into n skews and a scale, as given in equation 23.*

Proof: The proof will be developed in terms of matrices which are transpositions of skews, but the results extend to skew matrices (T_i in equation 24) by the relation:

$$A_1 \times A_2 \times A_3 \times \cdots = (\cdots \times A_3^T \times A_2^T \times A_1^T)^T$$

The example used in this proof will be shown for dimension 4, however the proof extends to arbitrary dimensions. A matrix X (not necessarily unimodular) may be represented as the product of matrices $Y_1, Y_2, \cdots$, which differ from the identity matrix in one column only as follows:

$$\begin{bmatrix} x_{11} & x_{12} & x_{13} & x_{14} \\ x_{21} & x_{22} & x_{23} & x_{24} \\ x_{31} & x_{32} & x_{33} & x_{34} \\ x_{41} & x_{42} & x_{43} & x_{44} \end{bmatrix} = \begin{bmatrix} y_{11} & 0 & 0 & 0 \\ y_{21} & 1 & 0 & 0 \\ y_{31} & 0 & 1 & 0 \\ y_{41} & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & y_{12} & 0 & 0 \\ 0 & y_{22} & 0 & 0 \\ 0 & y_{32} & 1 & 0 \\ 0 & y_{42} & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & y_{13} & 0 \\ 0 & 1 & y_{23} & 0 \\ 0 & 0 & y_{33} & 0 \\ 0 & 0 & y_{43} & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & y_{14} \\ 0 & 1 & 0 & y_{24} \\ 0 & 0 & 1 & y_{34} \\ 0 & 0 & 0 & y_{44} \end{bmatrix}$$

$$= \begin{bmatrix} y_{11} & y_{12}x_{11} & y_{13}x_{11} + y_{23}x_{12} & y_{14}x_{11} + y_{24}x_{12} + y_{34}x_{13} \\ y_{21} & y_{12}x_{21} + y_{22} & y_{13}x_{21} + y_{23}x_{22} & y_{14}x_{12} + y_{24}x_{22} + y_{34}x_{23} \\ y_{31} & y_{12}x_{31} + y_{32} & y_{13}x_{31} + y_{23}x_{32} + y_{33} & y_{14}x_{13} + y_{24}x_{32} + y_{34}x_{33} \\ y_{41} & y_{12}x_{41} + y_{42} & y_{13}x_{41} + y_{23}x_{42} + y_{43} & y_{14}x_{14} + y_{24}x_{42} + y_{34}x_{43} + y_{44} \end{bmatrix}$$

The right hand side is obtained by multiplying the Y matrices and simplifying. Comparing the corresponding elements, one may see that the $y_{i,j}$'s may be computed, given an X matrix with the restriction that the all square submatrices of X containing the element x_{11} have non-zero determinants. Any non-singular matrix may be row or column permuted to this form. By comparing elements, one sees that the first column of y's is equal to the first column of x's. The second column of y's is found by solving a system of one equation and one unknown, followed by back-substitutions. The third column of y's is found by solving a system of two equations and two unknowns, followed by back-substitutions. The fourth column requires solving a system of three equations and so forth. The restrictions on the square submatrices of

X containing the element x_{11} ensure that the systems of equations that need to be solved have solutions.

It can also be shown that the above sequence of products can be re-written as:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ \frac{y_{21}}{y_{11}} & 1 & 0 & 0 \\ \frac{y_{31}}{y_{11}} & 0 & 1 & 0 \\ \frac{y_{41}}{y_{11}} & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & \frac{y_{11}y_{12}}{y_{22}} & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & \frac{y_{32}}{y_{22}} & 1 & 0 \\ 0 & \frac{y_{42}}{y_{22}} & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & \frac{y_{11}y_{13}}{y_{33}} & 0 \\ 0 & 1 & \frac{y_{22}y_{23}}{y_{33}} & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \frac{y_{43}}{y_{33}} & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & \frac{y_{11}y_{14}}{y_{44}} \\ 0 & 1 & 0 & \frac{y_{22}y_{24}}{y_{44}} \\ 0 & 0 & 1 & \frac{y_{33}y_{34}}{y_{44}} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} y_{11} & 0 & 0 & 0 \\ 0 & y_{22} & 0 & 0 \\ 0 & 0 & y_{33} & 0 \\ 0 & 0 & 0 & y_{44} \end{bmatrix}$$

One can see that the scaling matrix can be "factored" out as long as none of the elements $y_{11}, y_{22}, \cdots$ are equal to zero. This is ensured by the fact that the determinant of X $(= y_{11} \times y_{22} \times \cdots)$ is not zero.

Thus, given a matrix X with the appropriate properties, a factorization into the type described above (equation 23) can be made. If $\prod_{i=1}^{n} y_{ii} = 1$ then then entire product is also unimodular.
□

Given the factorization of T in equation 23, the following properties are observed:

1. T is the transformation resulting from skewing $\mathcal{D}$ in each of its n dimensions followed by a volume preserving scale.

2. T is unimodular. $(\det T = \det T_1 \times \det T_2 \times \cdots \times \det T_n \times \det S = 1)$

3. The skew T_i only affects the i^{th} component of a vertex. All other components are left unchanged,

$$T_i x = T_i \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} x_0 \\ \vdots \\ x_{i-1} \\ t_{0i}x_0 + \cdots + t_{(i-1)i}x_{i-1} + x_i + t_{(i+1)i}x_{i+1} + \cdots + t_{ni}x_n \\ x_{i+1} \\ \vdots \\ x_n \end{bmatrix}$$

and thus T_i will only change the size s_i of the bounding box in the i^{th} dimension (see equations 21 and 22).

4. Since the elements t_{ij} in the skew transformation matrices are rational, a skew may map a point from an integral grid to a non-integral grid.

5. The scaling transformation S may be used to realign points onto integer gridlines which where misaligned by the skews.

One more important observation is that in each dimension i, there exist at least two vertices r_a and r_b (columns of R), $1 \leq a, b \leq k$ that determine the length of the i^{th} side.

$$\begin{aligned}
r_a &= \begin{bmatrix} r_{1a} \\ r_{2a} \\ \vdots \\ r_{na} \end{bmatrix}, \quad r_b = \begin{bmatrix} r_{1b} \\ r_{2b} \\ \vdots \\ r_{nb} \end{bmatrix} \\
u_i &= \max\{r_{i1}, r_{i2}, \cdots, r_{in}\} \\
&= r_{ia} = (r_a)_i \\
l_i &= \min\{r_{i1}, r_{i2}, \cdots, r_{in}\} \\
&= r_{ib} = (r_b)_i \\
s_i &= u_{i-1} - l_{i-1} + 1 \\
&= \max\{r_{ia} - r_{ib} + 1\} \\
&\quad \text{for all pairs of columns } (a, b) \text{ in } R \ : \ 1 \leq a, b \leq k
\end{aligned}$$

Theorem 3 *The skew transformation T_i which minimizes the size of a bounding box in the i^{th} dimension may be found by setting up the following linear programming problem. Minimize s'_i where:*

$$\left.\begin{array}{rcl} s'_i & \geq & T_i(r_a - r_b)_i + 1 \\ s'_i & \geq & T_i(r_b - r_a)_i + 1 \\ & \vdots & \end{array}\right\} \quad \text{for all pairs } (a, b) \ : \ 1 \leq a < b \leq k \qquad (25)$$

Proof: After transformation by T_i,

$$\begin{aligned}
s'_i &= \max_{0<a,b\leq k} \{r'_{ia} - r'_{ib} + 1\} \\
&= \max_{0<a,b\leq k} \{T_i(r_a)_i - T_i(r_b)_i + 1\} \\
&= \max_{0<a,b\leq k} \{T_i(r_a - r_b)_i + 1\}
\end{aligned}$$

where r_a and r_b are columns of ray matrix R. The minimum s'_i which satifies those constraints is $\max\{T_i(r_a - r_b)_i + 1\}$, which leads to the integer programming problem described above with the cost function $cost = s'_i$.
□

A dual optimization problem can be formulated using the duality theorem of linear programming [13] which does not need to generate all pairs of vertices to find the minimum transformed size.

The minimum bounding parallelpiped problem

Given a domain $\mathcal{D}$, find a unimodular matrix H and vectors l and u, which specify the parallelepiped $\mathcal{D}_h$ that bounds the domain $\mathcal{D}$ as follows:

$$\mathcal{D}_h = \{\ x \mid l \leq Hx \leq u\ \} \supseteq \mathcal{D}$$

and such that the volume of the parallelepiped $\mathcal{D}_h$ is minimized.

Theorem 4 *The minimum bounding parallelepiped problem is an equivalent problem to the minimum bounding box problem.*

Proof: Given a domain specified by $\mathcal{D} = \{x \ : \ Ax \geq b\}$, we can view a unimodular $n \times n$ transformation H as follows. Each row of H is a vector h_i which is normal to a pair of parallel hyperplanes defined by the equations $h_i x = l_i$ and $h_i x = u_i$ such that domain $\mathcal{D}$ is entirely contained between the hyperplanes. The scalars l_i and u_i are chosen such that the difference $u_i - l_i$ is minimal, which means that the hyperplanes bound domain $\mathcal{D}$ as tightly as possible. We call the parallelepiped whose sides are the n pairs of hyperplanes $\mathcal{D}_h$ and define it as follows:

$$\mathcal{D}_h = \{\ x \mid l \leq Hx \leq u\ \} \supseteq \mathcal{D}, \quad \text{where}$$

$$H = \begin{bmatrix} h_{11} & h_{12} & \cdots & h_{1n} \\ h_{21} & h_{22} & \cdots & h_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ h_{n1} & h_{n2} & \cdots & h_{nn} \end{bmatrix} = \begin{bmatrix} h_1 \\ \vdots \\ h_n \end{bmatrix} \qquad l = \begin{bmatrix} l_1 \\ l_2 \\ \vdots \\ l_n \end{bmatrix} \qquad u = \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{bmatrix}$$

The volume of $\mathcal{D}_h$ is:

$$\begin{aligned} Volume(\mathcal{D}_h) &= \frac{(u_1 - l_1) \times (u_2 - l_2) \times \cdots \times (u_n - l_n)}{\det(H)} \\ &= (u_1 - l_1) \times (u_2 - l_2) \times \cdots \times (u_n - l_n), \quad (\text{since } \det(H) = 1) \end{aligned}$$

Let $\mathcal{D}'_h$ be the domain obtained by computing $\mathcal{D}_h.H$ as follows:

$$\begin{aligned} \mathcal{D}'_h &= \{x' \mid x' = Hx,\ x \epsilon \mathcal{D}_h\}, \quad \det(H) = 1 \\ &= \{x' \mid l \leq x' \leq u\} \\ Volume(\mathcal{D}'_h) &= (u_1 - l_1) \times (u_2 - l_2) \times \cdots \times (u_n - l_n) \end{aligned}$$

Thus, $\mathcal{D}'_h = \mathcal{D}_h.H$ is the bounding box of the domain $\mathcal{D}.H$ (compare to equation 7), and hence the volume of the bounding box of $\mathcal{D}.H$ is the same as the volume of parallelepiped $\mathcal{D}_h$ defined by H. Thus to find H, we can formulate the following minimization problem:

$$\begin{aligned} V &= \min[(u_1 - l_1) \times (u_2 - l_2) \times \cdots \times (u_n - l_n)] \\ &\quad \text{such that :} \quad (\forall x \ :\ (Ax \geq b) \ \rightarrow \ l \leq Hx \leq u), \quad \det(H) = 1 \\ &= \min_{\det(H)=1} \prod_{i=1}^{n} [\ \max_{(Ap \leq b,\ Aq \leq b)} h_i(p - q)\ ] \end{aligned}$$

By the duality theorem of linear programming, the following is obtained:

$$V = \min_{\det(H)=1} \prod_{i=1}^{n} [\ \min_{\substack{(y_{i1}A = h_i,\ y_{i2}A = -h_i, \\ y_{i1} \geq 0,\ y_{i2} \geq 0)}} (y_{i1} + y_{i2})b\ ]$$

This is a non-linear optimization problem consisting of the product of linear optimization problems all under a common constraint that $\det H = 1$.

A resulting H matrix found by the solution of the above problem can be used as a transformation on $\mathcal{D}$ to minimize the volume of the bounding box. We also note that the order of the rows of H is unimportant, thus having found one matrix H which minimizes the bounding box, we can induce a set of equivilant solutions consisting of all row permuations of H. By performing row permutations, we can always find a solution H

which satisfies the conditions of theorem 2, and thus be factored into a set of skews.

$$H = \begin{bmatrix} h_1 \\ \vdots \\ h_n \end{bmatrix} = \prod_{i=n}^{1} T_i \quad , \quad T_i = \begin{bmatrix} 1 & 0 & \cdots & & 0 & 0 \\ 0 & 1 & & & 0 & 0 \\ \vdots & & \ddots & & \vdots & \\ t_{i1} & t_{i2} & \cdots & t_{ii} & \cdots & t_{in} \\ \vdots & & & & \ddots & \vdots \\ 0 & 0 & \cdots & & 0 & 1 \end{bmatrix}$$

$$h_i = t_i \times \prod_{j=i-1}^{1} T_j \quad , \quad t_i = \begin{bmatrix} t_{i1} & t_{i2} & \cdots & t_{in} \end{bmatrix}$$

$$\det(H) = 1 \quad \rightarrow \quad \prod_{i=1}^{n} t_{ii} = 1$$

The minimization problem above may now be reformulated in terms of the finding the sequence of skews which minimizes the bounding box.

$$V = \min_{(\prod_{i=1}^{n} t_{ii}=1)} \prod_{i=1}^{n} \left[\max_{(Ap \le b,\ Aq \le b)} t_i \prod_{j=i-1}^{1} T_j (p-q) \right]$$

Using the relation $\{Tp \mid Ap \le b\} \stackrel{Tp \rightarrow q}{=} \{q \mid AT^{-1}q \le b\} = \{p \mid AT^{-1}p \le b\}$ and the fact that $(\prod_{j=i-1}^{1} T_j)^{-1} = \prod_{j=1}^{i-1} T_j^{-1}$ the following formulation is obtained:

$$V = \min_{(\prod_{i=1}^{n} t_{ii}=1)} \prod_{i=1}^{n} \left[\max_{\substack{A'_i = A(\prod_{j=1}^{i-1} T_j^{-1}), \\ A'_i p \le b,\ A'_i q \le b}} t_i (p-q) \right]$$

$$= \min_{(\prod_{i=1}^{n} t_{ii}=1)} \prod_{i=1}^{n} \left[\max_{\substack{A'_i = \begin{cases} A \text{ when } i=1 \\ A'_{i-1} T_{i-1}^{-1} \end{cases}, \\ A'_i p \le b,\ A'_i q \le b}} t_i (p-q) \right]$$

By the duality theorem of linear programming, the following is obtained:

$$V = \min_{(\prod_{i=1}^{n} t_{ii}=1)} \prod_{i=1}^{n} \left[\min_{\substack{A'_i = \begin{cases} A \text{ when } i=1 \\ A'_{i-1} T_{i-1}^{-1} \end{cases}, \\ y_{i1} A' = t_i,\ y_{i2} A' = -t_i, \\ y_{i1} \ge 0,\ y_{i2} \ge 0}} (y_{i1} + y_{i2}) b \right]$$

We have shown that the skew formulation is equivalent to the hyperplane formulation of the problem.
□

Conjecture 5 *The volume of the transformed domain* $\mathcal{D}'_{\mathcal{BB}}$

$$volume(\mathcal{D}'_{\mathcal{BB}}) = s'_1 \times s'_2 \times \cdots \times s'_n$$

is globally minimized by sequentially minimizing the s_i *terms using the linear programming procedure described in theorem 3.*

Discussion: This proof is an open problem. A sketch of a proof is as follows. There is a unique minimum volume for every every instance of the minimum parallelepiped problem (and therefore for the minimum bounding box problem). Given a matrix H which is a solution, a series of skews can be found whose product is H (or a row permutation of H). The algorithm presented in this paper produces those skews. In changing the problem from computing hyperplanes to computing skews, the global condition $\det H = 0$ is transformed into computing the each skew, based on a previous skews. This is done by transforming the domain at each step by the newly computed skew. The fact that there is a class of solution matrices H (which are row permutations of each other), means that there is also a class of solution skew sequences all of which arrive at the same minimum volume. The 'greedy' nature of the algorithm does not lock us into local minimum as is usually the case in these kinds of problems, but instead leads to a unique solution from among the class of minimum skew sequence solutions.
□

5 Algorithm

Given $\mathcal{D}$, a domain of dimension n represented as a set of vertices, compute T by:
For $i = 1 \cdots n$ *do steps 1 through 4.*

1. Compute a list

$$E = \{e_{ij} \ : \ e_{ij} = r_i - r_j, \ \ r_i, \ r_j \text{ vertices of } \mathcal{D}, \ \ i < j\}$$

Given k vertices, there will be $k \times (k-1)$ elements in E.

2. Create a polyhedron consisting of the constraints:

$$\begin{aligned} s_i &\geq T_i \times e, \quad \text{for all } e \text{ in } E \\ s_i &\geq T_i \times (-e), \quad \text{for all } e \text{ in } E \\ &\vdots \end{aligned}$$

3. Using linear programming, find T_i so as to minimize s_i.

4. Replace $\mathcal{D}$ by $\mathcal{D}.T_i$
   Repeat for each dimension i

5. Compute the unimodular scale transformation S to make the resulting transformation T an integer matrix (if possible).

6. Compute $T = T_1 \times T_2 \times \cdots \times T_n \times S$

6 Scanning a Domain

Once a domain is allocated in memory using the methods described in this paper, it can be scanned relatively easily in the order it has been allocated by scanning the bounding box using nested DO-loops [2, 5, 4] with constant upper and lower bounds. The *lex*() function itself gives the order in which each element will be accessed. Elements of $\mathcal{D}_{BB}$ allocated in memory which do not belong to $\mathcal{D}$ can be filtered out by putting a domain membership test in the loop body. The indices `i, j, ...` could be tested against the equations which define the domain $\mathcal{D}$—however this can be costly in terms of run time. Another solution is that there could be a tag bit associated with each element which tells whether that element is a member of $\mathcal{D}$ or not. Such a procedure is illustrated in figure 5. This would be faster than a program which did a membership test using the domain constraints, but has the expense of a tag bit for each element. Also this tag bit would need to be computed and preinitialized by the compiler, adding to the compile time. An even faster method can be found by noticing that elements that are outside of the domain usually come in contiguous sequences, that is, there are usually several in a row. By skipping over all of a string of outside elements at once, the time spent processing elements outside of the domain could be further reduced. The unused value fields of outside elements could be used

```
a = D.base;
for (i=0; i<si; i++)
   for (j=0; j<sj; j++)
      ...
         {   if (*a.tag == outside) { a++; continue; }
             Compute(*a.value);  /* D[i,j,...] */
             a++;
         }
```

Figure 5: Scanning an allocated polyheron

and preprogrammed to store information needed to skip to the next good element or another table could be used to store skip information.

6.1 Scanning with a given schedule

Often, there is a timing function associated with a domain which dictates the order in which the domain is to be scanned. If this timing function is a full ordering of the points, then it can be used itself as a the *lex()* function and memory can be allocated in that order. However, if it is only a partial ordering, then we can use methods developed in this paper to create a full ordering *lex()* function which respects the timing function and reduces memory.

Assume that in a multidimensionsional domain, the domain has already been transformed to reflect the timing function in its first t (of n) dimensions. To respect the timing function, those dimensions cannot be skewed, or if they are skewed, loops must be written that scan those indices in the unskewed order. However, we can use our methods on the last $n - t$ dimensions to reduce the amount of memory required under the timing function constraint. Scanning techniques discussed above will work, as long as the timing dimensions are made the outer loops.

7 Conclusion

Memory allocation for domain based variables is required for efficient execution of recurrence equations. In this paper, we have demonstrated a method of allocating memory for arbitrary convex polytopes that involves finding a bounding box of the variable domain after being transformed by a unimodular matrix. The method we have described produces a memory allocation in which positions of variable elements in memory can be computed by an affine addressing function while at the same instant, attempting to reduce the amount of memory allocated to the variable. We have shown that the computation of the optimal transformation is a non-linear programming problem. Our method finds a transformation by solving a sequence of dependent linear programming problems. While we believe that the transformation produced by our method is the optimal one, the proof is still an open problem.

We have also shown methods to scan memory allocated by such a procedure. One such method with a tag bit for each element and a forwarding address for elements outside of the domain, is especially attractive in terms of execution time. Its has been shown how allocation and scanning may be done with repect to a given timing function.

The methods introduced in this paper are useful in the compilation of recurrence equations into efficient executable code.

References

[1] A. V. Aho and J. D. Ullman. *Principles of Compiler Design.* Addison-Wesley, Reading, Mass., 1977.

[2] C. Ancourt and F. Irigoin. Scanning polyhedra with DO loops. In *Third Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 39–50, ACM SIGPLAN, ACM Press, 1991.

[3] R. S. Bird. Tabulation techniques for recursive programs. *Computing Surveys*, 12(4):403–419, Dec 1980.

[4] Zbigniew Chamski. *Environnement logiciel de programmation d'un accélérateur de calcul parallèle.* PhD thesis, l'Université de Rennes I, Rennes, France, Feb 1993.

[5] Zbigniew Chamski. Scanning polyhedra with DO loop sequences. In *Workshop on Parallel Algorithms*, Sophia, Bulgaria, August 1992.

[6] M.C. Chen. *Transformations of Parallel Programs in Crystal.* Technical Report YALEU/DCS/RR-469, Yale University, 1986.

[7] F. Fernández and P. Quinton. *Extension of Chernikova's Algorithm for Solving General Mixed Linear Programming Problems.* Technical Report 437, IRISA, Oct 1988.

[8] R. M. Keller and M. R. Sleep. Applicative caching. *ACM Transactions on Programming Languages and Systems*, 8(1):88–108, January 1986.

[9] H. Le Verge, C. Mauras, and P. Quinton. The ALPHA language and its use for the design of systolic arrays. *Journal of VLSI Signal Processing*, 3(3):173–182, September 1991.

[10] Christophe Mauras. *Alpha, un langage équationnel pour la conception et la programmation d'architectures parallèles synchrones.* PhD thesis, l'Université de Rennes I, Rennes, France, Dec 1989.

[11] Christophe Mauras, Patrice Quinton, Sanjay V. Rajopadhye, and Yannick Saouter. Scheduling affine parameterized recurrences by means of variable dependent timing functions. In S. Y. Kung and E. Swartzlander, editors, *International Conference on Application Specific Array Processing*, pages 100–110, IEEE Computer Society, Princeton, New Jersey, Sept 1990.

[12] Simon Peyton Jones. *The Implementation of Functional Programming Languages. PHI Series in Computer Science, (editor, Hoare, C. A. R.)*, Prentice Hall, 1987.

[13] A. Schrijver. *Theory of Integer and Linear Programming.* John Wiley and Sons, 1986.

[14] M. Wolfe. *Optimizing Supercompilers for Supercomputers.* MIT Press, Cambridge, Mass., 1989.

Unité de recherche INRIA Lorraine, Technôpole de Nancy-Brabois, Campus scientifique,
615 rue de Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399